Znews Vol. 1 No. 1, March 1989

Fumble Recovered: First and Ten

Although the fullback fumbled the handoff from Zedcor, we are finally able to bring you the first issue of *Znews*! It appears that Federal Express lost the subscriber list and some other goodies that the boys up at the big house mailed to us; it took about two weeks to get our signals straight thereafter. In truth, a lot of our mail gets rerouted to Arkansas. UPS and FedEX seem to get AK and AR mixed up a lot. I bet somebody down in the Ozarks is wondering why they got so much software from that outfit in Arizona...

Murphy's laws are immutable, I guess.

Back at the ranch, I wanted to take a few paragraphs this issue and get all of our initial announcements out of the way. Scores of you have had some very interesting questions.

First, in no way shape or form are we a subsidiary of Zedcor, Inc. We share no management in common. My company, Ariel Publishing, is an independent publishing house. The story of our acquisition of the ZBasic newsletter is short and sweet. All of us here at Ariel Publishing are unabashed fans of ZBasic and we wanted to see a regular monthly newsletter supporting the language. On the spur of the moment late last December, I called the head Z-man, Mike Gariepy, and asked if he'd be interested in turning over the newsletter operations to us. He was, and as a ZBasic programmer I'm thankful for it. Part of his motivation was that our situation as a publisher enabled us to deliver more newsletter to you more often. That is good for you and for ZBasic. It was a win-win situation, I think, and I hope you'll agree as the months go by.

Second, we received a flurry of orders from some of you renewing your subscriptions at our special early bird rate. I thank you! As you might imagine, though, there are two sets of records for you folks - Zedcor's and ours. To help insure accuracy as we begin merging the two, watch your mailing labels ***starting with the April issue.***

The mailing labels will reveal your expiration date and whether or not you've ordered the quarterly disk. A typical label will look something like this:

```
Mar 90      DISK MS-DOS

Zebulin ZFan
1234 Zoroaster Drive
Zygote, AZ 66509
```

Our friend Zebulin, here, will receive his last issue in March of 1990. He has also ordered the MS-DOS version of the quarterly disk.

- If you have NOT ordered the quarterly disk, the right hand side of the information line on your labels will be blank.

- If you have NOT yet renewed your subscription, your expiration date will be the same as that which Zedcor passed on to us. If it is wrong, call us and we'll fix it up for you.

- If you have made an early renewal with us, your expiration date should include the number of months left on your old subscription PLUS the number of issues on your new subscription.

If you find any mistakes or have any questions, please drop us a line or give us a call. We guarantee your satisfaction, and that includes our subscription services (gulp!). Just remember (for Zedcor's sake) that Ariel Publishing is now handling anything to do with the newsletter.

Speaking of the quarterly disk, I just wanted to mention that the first one will be mailed out with the June issue. And speaking of issues, we'll be appearing in your mail boxes mid-monthly from now through the summer.

I hope to be able to push production forward to the beginning of each month at that point.

The early renewal prices are $29.95 for one year and $52 for two (the regular rate is $35 for one year, $65 for two). The early bird offer is set to expire April 1, 1989. Due to the delay in getting this issue to you, we'll extend the offer to April 15, 1989. Maybe you can prespend a little of your tax refund.

And don't forget, no matter when (or if) you renew, we will be fulfilling your old Zedcor subscriptions *completely* and at no charge to you. We'll be doing this on an issue-for-issue basis, though, so in terms of time your subscription will run out considerably faster (they were a quarterly, we are a monthly).

I am pleased to report that virtually ALL of you who commented about our potential content have had the same sorts of ideas we have. We'll be focusing on ZBasic DOS utilities, file conversion/creation routines for popular packages like AppleWorks, Lotus, etc., charting functions, BTree routines, etc. Let me make one thing perfectly clear: ***Znews*** *is a programmer's newsletter. An overwhelmingly large proportion of our articles will contain source code. We don't expect that you are all professionals by any means, but we do believe you want information about and examples of ZBasic source code. We will NOT be reviewing hardware; that is Byte's market.*

Don't forget that this is YOUR newsletter, so if you have something you'd like to see (or write!) give us a call or drop us a line.

Incidentally, if you'd like a copy of our writer's guidelines, write or call. We usually pay between $25-$75 for an article, although that is negotiable depending on the length and quality of the submission. You disk subscribers will have the guidelines automatically - they'll come on the first disk.

Another note - those of you who paid for your subscriptions with credit cards should be aware that your bill might show that you made your purchase from Teachers' Software Company. Please don't do a "chargeback". That is really us. When I first jumped into the computer biz, I wrote education related programs. My bank is none too good at name changes.

Finally, an amazingly large number of you have wanted to know about Unalakleet. Well, it is pronounced pretty phonetically, YOU-nah-la-kleet. It is Inupiaq eskimo for "mouth of the river". We moved here originally because my wife and I both taught at the local school. Yes it is beautiful, in a stark Bering Strait sort of way, and the fishing is incredible. I will miss it when we move at the end of May. That is a sort of an announcement, too. We'll keep you informed as to our new address, etc.

Super Input: A Line Editor+

Since ZBasic generates machine code native to your particular computer, most any ol' operation can be done at incredible rates of speed. This is most noticeably true with string operations, especially when compared to interpreted BASICs. Such a facility with words and letters enables ZBasic programmers to develop editing routines previously impossible except in assembly language. But as I'm sure you all can imagine, detailed string manipulations in assembly can be a pain to write. Thankfully, the Z-Gang took care of that for us.

That's not to say that writing a full blown text editor in any language would be easy. I've struggled with text editors in a variety of environments on several different machines. It is tricky. But I think that we can examine some of the considerations involved so that you can write your own routines without the pain and bloodletting I had to endure. Today's program is a one line input long function that we'll eventually expand into a rudimentary full screen oriented text editor. I've kept it as generic as possible, so I believe it will work in any version of ZBasic.

One of the features that I insist upon is the ability to customize the cursor. We shall define our initial cursor to be CHR$(127). On most computers, that's the checkerboard. I also want to be able to adjust the blinkrate of our cursor. We shall define a variable, BLINKRATE, to take care of that. We'll flash the cursor after executing a timing loop, hence the larger the BLINKRATE variable, the slower the cursor will actually flash. Keep in mind that the rate of cursor flash is also hardware dependent. The speedier the CPU, the faster the flicker.

Speaking of the timing loop, we also need to keep track of whether the cursor is currently being displayed or the character under the cursor. That flag will be called, strangely enough, CURSORFLAG.

One of the reasons I want to control the cursor character is that I also want an insert mode. In the early going here we'll also initialize the flag that reveals our current mode, INSERTFLAG.

Incidentally, some folks might wonder why I defined constants like BS$ and RIGHTARROW$ when I could have just used their ASCII values throughout the program. The answer is that it allows for easy customization. You can remap all of our special characters (for example, BELL$) by changing their definitions. You won't have to touch another line of code thereafter.

In this vein, the maximum length of the input is also defined here in the early going. Since this is a line oriented editor at present, I made the default equal to one screen line of 80 characters. You can redefine the maximum length however you want; you can even make it a parameter to be passed to the function.

```
 REM ------------------------------------
 REM Super Input
 REM  by Ross W. Lambert, Znews Editor
 REM
 REM This program is in the public domain
 REM
 REM config info: space req'd between
 REM keywords, default variable is
 REM integer, expressions optimized to
 REM integer
 REM ------------------------------------
 :
 DIM 1 BELL$, BS$, SP$, CURSOR$, CR$, 2 C$: REM save some memory by limiting
string lengths
 DIM 80 TR$, TL$, LN$   : REM 80 chars max input (adjust to suit)
 :
 BELL$ = CHR$(7) : REM a beep on most computers
 BS$ = CHR$(8) : REM the backspace (usually a back arrow)
 DELETE$ = CHR$(127) : REM delete key
 CURSPOS = 1   : REM cursor starts in 1st position
 BLINKRATE = 110 : REM controls rate of cursor flash (lower means faster)
 SP$ = CHR$(32) : REM a space
 CR$ = CHR$(13) : REM a CR
 RIGHTARROW$ = CHR$(21) : REM right arrow
 ESC$ = CHR$(27) : REM escape key
 INSERTFLAG = 0 : REM start in overwrite mode (1 = insert mode)
 CURSORFLAG = 1 : REM start with cursor showing
 CURSOR$ = CHR$(127) : REM start with a checkerboard cursor
 MAXLEN = 80     : REM max input length - adjust to suit
```

```
:
:
LONG FN SUPER_INPUT$ (X,Y): REM pass starting coordinates
  LN$ = ""                  : REM init input line variable
  CLS PAGE  : REM clear window from cursor down (establish your own edit
window)
  C$ = "" : REM clear character string
```

After the initialization is out of the way, we must tackle our input loop and work with the display of the cursor.

First, we have to determine where the cursor is. After all, if no characters have been entered or the user has used a right arrow key to move out beyond the last letter, there is no character to exchange with the cursor. We test for this by comparing the current cursor position with the length of the input line. If the cursor is beyond the last letter, just exchange the cursor with a space.

Our next task, at long last, is to gather the actual keypresses. This we shall do with a DO/UNTIL loop. The first statement inside the loop is a call to the INKEY$ function. Immediately thereafter, we check the state of CURSORFLAG. If it is not equal to zero, we print the cursor and then immediately backspace so we do not advance down the line. If CURSORFLAG equals one, then we print the character under the cursor followed by a backspace.

The main point to remember about INKEY$ is that it just scans your keyboard queue once. If no key is down, it passes control onto the next statement with nary a glitch (though TRON can muck up the works). If no key is down, then, we increment the variable BLINK. When BLINK equals the BLINKRATE, we exchange the cursor with the letter under it (and vica versa the next time around).

```
  :
  "GetKey"
  IF CURSPOS <= LEN (LN$) THEN CHAR$ = MID$ (LN$,CURSPOS,1) ELSE CHAR$ =
SP$
    LOCATE X + CURSPOS - 1,Y : REM position cursor
    DO
      C$ = INKEY$
      IF CURSORFLAG THEN PRINT CURSOR$;BS$; ELSE PRINT CHAR$;BS$;
      LONG IF LEN(C$) = 0
        BLINK = BLINK + 1
        LONG IF BLINK = BLINKRATE
          BLINK = 0    : REM reset counter
          IF CURSORFLAG THEN CURSORFLAG = 0 ELSE CURSORFLAG = 1
        END IF
      END IF
    UNTIL LEN(C$) : REM loop until a key is pressed
    :
    WLEN = LEN (LN$) : REM get current length of input line
    PRINT CHAR$;BS$; : REM replace character under cursor before moving on
    :
```

After grabbing a character, our next job is to analyze it. The first thing to check is to see if it is our command for changing modes, CTRL-E. If so, just set a flag, change the cursor character, and skip town. Like most good Americans, we'll deal with the consequences our actions later.

Incidentally, if you ever decide to add functions to this routine, this is probably the best spot to put them in, especially if you use control characters.

```
LONG IF C$ = CHR$(5) :REM exchange mode command (CTRL-E)
  IF CURSOR$ = CHR$(127) THEN CURSOR$ = CHR$(43) ELSE CURSOR$ = CHR$(127)
  IF INSERTFLAG = 0 THEN INSERTFLAG = 1 ELSE INSERTFLAG = 0
  GOTO "GetKey"
END IF
:
```

Even if the character we snatched from the keyboard is not a control character, that does not mean it should be added to our input line. There are several special characters which deserve special treatment. They are the backarrow, the rightarrow, the delete key, the escape key, and the return key (they might be named differently on your computer).

To avoid needless comparisons (which slow down the process), we'll check our character first and if it is not one of our "specials", we'll branch to the normal text character handler. In fact, although it is generally not a good idea to exit structured loops early (DO/UNTILs, WHILE/WENDs and the like), I have used the GOTO profusely to take us back to the input loop as soon as possible. Even in a compiled language, it makes sense to consider speed. I think a quick GOTO is appropriate for this program. After all, your input routine might encounter a very speedy typist someday.

Return will end the whole process, accepting the input "as is". We exit after one of those.

For a backspace we must check the cursor position again. If we are as far left as we can be, we gotta beep our displeasure. Otherwise decrement the CURSORPOS variable and move backwards.

The same sorts of rules apply to the right arrow. If we've gone as far to the right as we can, beep. Otherwise increment the CURSORPOS variable. Again, we must check to see if we're hanging out past the end of text or sitting on a character.

The delete key is a special case. I prefer a delete key that always does what its name implies. And I like it to eradicate the character to the left of the cursor. Like the backspace, we beep if we cannot go any further to the left. Otherwise, split the input line into two halves. The left half stops immediately *before* the character we wanna kill. The right half spans the rest of the line from the cursor to the end. Glue the new pieces together again, decrement the cursor position, and go back for another key. Oh yes, don't forget to print out the changed line, though, or your user will get awfully confused.

Handling the escape key depends on what you want it to do. I prefer to use it like an escape hatch; when pressed the user is allowed to back out into a menu or some earlier activity. The problem is that the body of my program needs to know when that occurs. Since long function programming is designed to create fairly self-contained modules, that can be a little bit of a trick. I chose to handle it by making LN$ = ESC$ and then exiting. The calling routine must then check the value of LN$ before proceeding. If escape has been pressed, it then acts accordingly.

```
REM handle special chars (backarrow, rightarrow, delete, esc, cr)
:
IF C$ => SP$ AND C$ < CHR$(127) THEN "Normal_Char"
```

```
    IF C$ = CR$ THEN "Exit"
    :
    LONG IF C$ = BS$  : REM key press was a backspace
      IF CURSPOS = 1 THEN PRINT BELL$;:GOTO "GetKey"
      CURSPOS = CURSPOS - 1
    END IF
    :
    :
    LONG IF C$ = DELETE$      : REM delete key pressed
      IF CURSPOS = 1 THEN PRINT BELL$;:GOTO "GetKey"
      CURSPOS = CURSPOS - 1
      TL$ = LEFT$ (LN$,CURSPOS - 1) : REM chop string left
      TR$ = RIGHT$(LN$, WLEN - CURSPOS)
      LN$ = TL$ + TR$
      PRINT BS$;TR$;
      CLS LINE : REM clear to end of line to erase trail
    END IF
    :
    LONG IF C$ = RIGHTARROW$   :REM right arrow
      IF CURSPOS = MAXLEN THEN PRINT BELL$;:GOTO "GetKey": REM can't go past
maxlen
      LONG IF CURSPOS > WLEN :REM past text?
        PRINT SP$;
        LN$ = LN$ + SP$
      XELSE
        PRINT CHAR$;  : REM mid text
      END IF
      CURSPOS = CURSPOS + 1
    END IF
    :
    IF C$ = ESC$ THEN LN$ = ESC$ : GOTO "Exit"
    :
    GOTO "GetKey"    : REM ignore illegal or meaningless chars
    :
```

At long last we're ready to add our text to the input string. Unfortunately, it is not quite as easy as just concatenating the character to the existing string. We are constrained by the mode and the position of the cursor.

If we're in insert mode, we gotta insert the character at the present cursor position and shove everything else to the right. That is accomplished by dividing the line in two, as before, but plopping the new character down in between the two halves and gluing the whole mess back together.

If we're in overwrite mode, we do the same sort of thing, except that the left half of the string, TL$, does not contain the character under the cursor (somewhat reminiscent of the way we handled delete). Then we plop in our text character.

Regardless of the mode, if the cursor is past the end of the line, then and only then do we have the luxury of just tacking it the character on to the end of the input string.

```
    "Normal_Char"
    :
```

```
LONG IF CURSPOS <= WLEN      : REM are we mid text?
  LONG IF INSERTFLAG = 0       : REM overwrite mode?
    TL$ = LEFT$(LN$,CURSPOS-1)   : REM chop string left...
    TR$ = RIGHT$(LN$,WLEN - CURSPOS) : REM and right
    LN$ = TL$ + C$ + TR$
    CURSPOS = CURSPOS + 1
    IF CURSPOS > MAXLEN THEN CURSPOS = MAXLEN
    PRINT C$;TR$;
    GOTO "GetKey"
  END IF
  :
  LONG IF INSERTFLAG = 1      :REM insert mode
    IF WLEN = MAXLEN THEN PRINT BELL$;:GOTO "GetKey"
    TL$ = LEFT$(LN$,CURSPOS - 1)
    TR$ = RIGHT$(LN$,WLEN - (CURSPOS - 1))
    TR$ = C$ + TR$
    LN$ = TL$ + TR$
    CURSPOS = CURSPOS + 1
    LOCATE X,Y
    PRINT LN$;
    LOCATE X + CURSPOS - 1,Y
    GOTO "GetKey"
  END IF
END IF
:
IF WLEN => MAXLEN THEN CURSPOS = MAXLEN : PRINT BELL$;:GOTO "GetKey"
CURSPOS = CURSPOS + 1       : REM here only if cursor past text
LN$ = LN$ + C$
PRINT C$;
IF CURSPOS > MAXLEN THEN CURSPOS = MAXLEN  : REM cursor hits wall
GOTO "GetKey"
:
  "Exit"
END FN = LN$
:
REM quick demo
:
CLS
PRINT@(0,5)"Type away:"
MYSTRING$ = FN SUPER_INPUT$ (0,7)
IF MYSTRING$ = ESC$ THEN MYSTRING$ = "You pressed ESCape!"
PRINT@(0,10);MYSTRING$
END
```

Notice how I handled a situation wherein the cursor is at the end of the text, but has reached the maximum length of the input. There are many possible ways to deal with such things, but I chose to accept any character typed and overwrite the last character in the string. In practice, it is kind of like the cursor hit a brick wall. It is fairly intuitive, though.

Thus far, our editing function is really just an improved LINEINPUT. But throw in a few INDEX array strings, the up and down arrows, some more parameter checking, and a little garlic, and I think you'll be surprised at what comes out.

But that's next month.

AWP Anatomy

I deliberately left the word "Apple" out of the headline for this article because I didn't want to lose all of you MS-DOS and Z-80 folks right off the bat. I will be focusing on Apple II's and AppleWorks word processor files, but the principles apply to virtually ANY version of ZBasic and ANY foreign data file. On top of that, familiarity with AWP file anatomy might come in handy for you someday. I know lots of folks who work with IBMs or clones at their place of employment and then do their "homework" with an Apple II. This article might be a guide for those of you who need to mess around with an AWP file, whatever your motivation or computer of choice.

Furthermore, I am going to introduce several stylistic elements which may help to standardize the plethora of source code formats we see in any version of ZBasic. So stay tuned.

Although the program we'll eventually work through converts an AWP file to a standard text file, our real agenda is to pick apart the AWP file format. Nevertheless, there is a real use for today's code: the TXT files created by AppleWorks have carriage returns appended to the end of every single line. This makes many other programs believe that your text file consists of zillions of very short paragraphs. This can be problematic, at the very least, and a royal pain in the chips on occassion.

Our program will automatically convert an AWP file to an "unformatted" TXT file, i.e. a file that only has carriage returns at the end of each paragraph. There are commercial packages available that will accomplish the same task (most notably *TimeOut: Power Pack* from Beagle Bros), but our code is educational and free (well, sort of).

AppleWorks word processor files are of the ProDOS filetype $1A (26 decimal). All files of this type are split up into two distinct parts, the header and the line records.

The header is a 300 byte data area at the beginning of every AWP file. It does not contain any pertinent information for our present purposes, but it may interest you to know that it does contain tab stop values, pagination flags, mail merge data, and a fifty byte area available for your own use. This fifty byte gap begins at byte 250 and runs to the end of the header, byte 299. (Remember that computer types always start counting with zero, so byte 299 is really the 300th and last byte of the header.)

The line records are a series of flags and data bytes which contain information about the text portion of the file. Each line record has as its first two bytes a pair of one byte flags. Robert Lissner, the creator of AppleWorks, labels byte 0 of each line record +000 and byte 1 +001. I shall adhere to his conventions.

Interestingly enough, it is the second byte, i.e. +001, that carries that most important information. If byte +001 equals 208, then that tells us that the current line consists of nothing more than a carriage return. In such instances, byte +000 contains the horizontal position of the carriage return. Notice how such an arrangment can save many bytes of memory; instead of storing a bunch of spaces plus a CR in memory, AppleWorks merely stashes away two bytes.

If byte +001 is greater than 208, the current line record holds a formatting command. This includes all of the commands available when you press OPEN-APPLE-O, including platen width, lines per inch, margins, and so forth. In these instances, byte +001 contains the code for the formatting command, and byte +000 stores any additional data required for that command. This additional data might be the new margin width, a page number, etc.

If byte +001 is equal to zero, we have finally struck paydirt - that signifies text on a line. The next few bytes are not text data, though.

We bump into a couple more flags.

Line records which contain text have two more flag bytes - +002 and +003. Byte +002 holds the screen column for the first text character in the line. If we were going to create a formatted text file, this might be significant, but we want a "raw" text file. We can ignore byte +002 at present.

Byte +003 is quite important, however. If it is greater than 128, then there is a carriage return at the end of the line. This particular carriage return is important to us; it marks the end of a paragraph. In this instance, the number of bytes of actual text to follow is equal to the value in byte +003 minus 128.

If there is no carriage return, the number of text bytes to follow is equal to the value in +003. For those of you who dabble in assembly language, the number of text bytes to follow is held in the lower seven bits of byte +003.

Once we have identified some actual text, we can scan it and transfer it to our unformatted text file with no real difficulty. The only caveat is that there are eleven control characters which AppleWorks embeds into the text. These are also related to formattng features. Their codes and meaning are outlined below:

AppleWorks Embedded Control Codes

Value in decimal	Meaning
01	boldface begin
02	boldface end
03	superscript begin
04	superscript end
05	subscript begin
06	subscript end
07	underline begin
08	underline end
09	print page number
10	enter keyboard - get keyboard input during printing
11	sticky space

I have found these embedded codes quite useful. In my AppleWorks to PostScript translator, for example, the first eight codes signal font changes (since boldface, italic and friends are, in practice, different fonts on the LaserWriter). In this manner, I can actually typeset text right in AppleWorks. After changing fonts, I discard the code. Although we will ignore these codes altogether in our text file translator, keep in mind that they might be useful to you later.

Now that we know the anatomy of an AWP file, we must stop and make some plans. We must design our own program.

First and foremost, we've got to figure out a way to get at the information and data in the AppleWorks file in the fastest manner possible. Since disk access slows down the entire process, we want to read in as much of our AWP file as we can. If you make this routine a part of a larger program (or use the 64K version of ZBasic), you'll want to compile your code, break out, and then do a MEM command to see how much memory space you have left available for a buffer. I arbitrarily chose to create a 6K buffer.

We shall use a favorite ZBasic trick to create our buffer - DIMension an integer array such that it creates the space we need and then use VARPTR to grab the address of the zeroth element. Remember that each element in an integer array contains two bytes, thus we need only DIM our array to hold half as many elements as bytes. For example, if we needed 1000 bytes, we'd DIMension our array to hold 500 elements.

In the interest of brevity, this program is pretty brutal towards the user. If you plan on using it regularly, I adjure you to put in a catalog routine, better error handling, and better editing features. It could drive you stark raving mad elsewise.

```
REM ------------------------------------------
REM ZBasic AWP File Reader
REM   by Ross W. Lambert, Editor
REM   Znews
REM
REM This routine is in the public domain
REM
REM configuration info: default variable is
REM integer, arithmetic expressions optimized
REM to integer, 128K version used, spaces
REM required between keywords, double and
REM single precision variables configured to
REM six significant digits or better
REM ------------------------------------------
:
:
DIM DATABUF%(3072) : REM create 6K array
DIM 64 AWPPATH$, TXTPATH$, FILE$ : REM longest filename is 64 chars long
DIM 1 CR$, INV$, NORM$
:
INV$ = CHR$(15)   : REM create inverse text
NORM$ = CHR$(14)  : REM normal text
CR$ = CHR$(13)    : REM carriage return
AMOUNT = 6144    : REM size of data buffer (adjust to suit)
:
:
```

I've done two things of significance thus far that I would like to see in all ZBasic listings. First, I outlined pertinent configuration details in the listing header. Second, I created my program around an "outline" which follows this general format: 1) The header - which should include the name and address of the author (if you're willing), copyright information, and configuration expectations. 2) DIM statements - these MUST be the first lines of code in any program which must use CHAINing, and it is not a bad idea to do this first, anyway. 3)

Declare constants - this helps to make programs self documenting. Your fellow programmers will be able to know the purpose and meaning of each constant right from the start. 4) All function definitions - grouping them in one place before you begin the body of your code is necessary, anyway. 5) Program initialization - setup the screen, get pathnames, etc. 6) Program main - the main logic. And finally, 7) All subroutines.

I do enjoy the freedom inherent with BASICs in general and ZBasic in particular. As I have moved into the professional programming ranks, however, I have become increasingly aware that at least some elements of structure save time. I'll avoid a long dissertation here, but I consider the above elements the first steps towards helping us all write better software.

Moving right along, we come to the next issue; how do we get the AWP file into memory? ZBasic's built in I/O routines work best when you are working with known quantities and you know what you're READing. They can be twisted and squeezed to do our present job, too, but it is much more straightforward to make use of the ProDOS MLI, the Machine Language Interface (shudder). If you don't like assembly language, don't choke here - ZBasic provides a fairly painless interface. The routines we present will work for any ol' file, too, so you don't really need to know any assembly to get the job done.

Those of you who've explored your Apple II ZBasic disk are probably wondering why I did not use the BLOAD.FN they supplied. The main reason is that it will only BLOAD a file that will fit entirely into memory all on one read. There is no built in mechanism for pulling in parts of a long file. As you may surmise, with the advent of the GS and memory expansion cards, there are lots of *long* AWP files floating around. Many of mine are in excess of 100K! We therefore need a way to pull them in a section at a time. So let us proceed...

We'll leave a detailed explanation of the MLI to Professor Gary Morrison (next month). For now, keep in mind that we will be dealing with TWO different kinds of "file numbers". As you all know, ZBasic wants you to number an OPEN file. Likewise, the ProDOS MLI. We ought not get the two confused.

Like ZBasic, the ProDOS MLI requires that a file be OPENed before any more file related actions can happen. Thus our first long function OPENs our AppleWorks file. Note that you must pass it the name of the file AND the ZBasic file number. This is because all I/O operations require a 1K buffer. ZBasic builds these at $8900 on down in the Apple's main memory, 1K for each open file. The OPEN_FILE function DOES return a value, that of the *ProDOS* reference number for the file. That will be important to us later.

```
LONG FN OPEN_FILE (FILE$,FNUM)
  BUFFER% = &8900 - (FNUM * &400)      :REM get 1K I/O buffer for ProDOS
  POKE &1F00,3                         :REM three parms for this call
  POKE WORD &1F01, VARPTR (FILE$)      :REM pass pointer to filename
  POKE WORD &1F03, BUFFER%             :REM tell ProDOS where buffer is
  MACHLG &A9, &C8, &20, &0865          :REM Open the file
  REF_NUM = PEEK (&1F05)               :REM Get ProDOS reference number
END FN = REF_NUM
:
```

Our next long function will find out how long our AppleWorks file really is (or any other file, for that matter). You must pass this function the ProDOS reference number we referred to earlier. It will return the file length.

```
:
```

```
LONG FN GET_EOF# (REF_NUM)
  POKE &1F00,2                    :REM two parms for this call
  POKE &1F01, REF_NUM
  MACHLG &A9, &D1, &20, &0865 :   REM make the call
  FILE_LEN# = PEEK WORD (&1F02) + PEEK (&1F04) * 256 : REM length of file
END FN = FILE_LEN#
:
```

The next logical step is to READ in our file, or at least as much of it as will fit in our buffer. Our READ_FILE function needs the ProDOS reference number, the address of the data buffer, and the amount of the file we want to READ. It, too, returns a value - the number of bytes actually sucked in. If you request to READ more bytes than are in the file, you won't generate an error, you'll just get the whole file. Who said computers are unforgiving?

Incidentally, the ACTUAL_READ variable can never exceed 6144 (bytes) in our program because we never request more than that. You may need to watch the variable type in your own applications.

```
:
LONG FN READ_FILE (REF_NUM, ADRS, REQUEST)
  POKE &1F00, 4                   :REM 4 parms for read
  POKE &1F01, REF_NUM             :REM pass ref num of file
  POKE WORD &1F02, ADRS           :REM where to put data?
  POKE WORD &1F04, REQUEST        :REM how much of file do we get?
  MACHLG &A9, &CA, &20, &0865     :REM Read the file into memory
  ACTUAL_READ = PEEK WORD (&1F06) :REM how much did we actually get?
END FN = ACTUAL_READ
:
:
```

We also need to CLOSE any OPEN files. You can do this one at a time by specifiying the ProDOS reference number, or you can just pass this function a zero and it will close everything in sight.

```
:
LONG FN CLOSE_FILE (REF_NUM)
  POKE &1F00, 1 : REM one parm
  POKE &1F01,REF_NUM : REM reference number
  MACHLG &A9, &CC, &20, &0865 : REM Make sure the file is closed
END FN
:
```

The next function is the trickiest beast of all, the SET_MARK routine. This serpent-like bugger allows us all the flexibility in the world. If you don't want to read a file in from the first byte, just set the MARK to whatever byte you want to begin with. This function and the length parameter of the READ_FILE function allows us to arbitrarily pick any chunk of any file and lay it right into memory, smooth as silk. SET_MARK requires the ProDOS reference number and your target byte (an offset from byte zero of the file).

```
LONG FN SET_MARK (REF_NUM, NEW_MARK#)
```

```
   POKE &1F00,2    : REM two parms
   POKE &1F01,REF_NUM : REM ProDOS reference number
   BYTE2 = INT (NEW_MARK#/65536) :REM divide potential three byte # into
parts
   BYTE1 = INT (NEW_MARK# - BYTE2 * 65536) / 256
   BYTE0 = NEW_MARK# - (BYTE2 * 65536) - (BYTE1 * 256)
   :
   REM now do pokes
   POKE &1F02,BYTE0
   POKE &1F03,BYTE1
   POKE &1F04,BYTE2
   :
   MACHLG &A9, &CE, &20, &0865    : REM set new mark
 END FN
 :
 :
```

You may be wondering why the FILE_LEN# and NEW_MARK# variables are double precision floating point and not integer. The reason is that ProDOS 8 allows file lengths of up to 16MB. Needless to say, that exceeds the 65535 limit of your regular everyday two byte integer variable. Such large numbers are really three byte integers and are divided up into memory with the lowest byte first - as shown in the NEW_MARK function.

Our program initialization is next, doing nothing fancy except using the SET_MARK function to let us skip past the 300 byte AWP header.

```
 REM ------------------------
 REM Start of Program
 REM ------------------------
 :
 ON ERROR GOSUB "DiskError"
 :
 BUFFERSTART  = VARPTR (DATABUF%(0)) : REM find beginning of buffer
 BUFFEREND  = BUFFERSTART  + AMOUNT
 :
 MODE 2
 PRINT INV$;" AWP to TXT File Converter ";NORM$
 PRINT@(0,4)"Name of AWP file: ";
 INPUT AWPPATH$
 PRINT@(0,6)"Name of TXT file: ";
 INPUT TXTPATH$
 :
 OUR_REF = FN OPEN_FILE (AWPPATH$,1) : REM open AWP file
 AWPLEN# = FN GET_EOF# (OUR_REF) : REM find out how long it is
 TOTALREAD# = 300 : REM we're starting 300 bytes into file to skip header
 FN SET_MARK (OUR_REF,300) : REM move file pointer past header
 GOSUB "DoRead"
 :
```

I have used the variable AMOUNT to set up the size of our AWP file buffer. In our case, AMOUNT will never exceed 6K (6144 bytes), so a normal integer variable is appropriate. If you need to change the buffer size, just change AMOUNT in the constant definition section.

At this point I define a variable called BYTECOUNT that keeps track of our current position in memory. In a few lines we will be scanning the AWP file and interpreting the line record flags in order to dig out the actual text data. Shortly after BYTECOUNT is defined, we use standard ZBasic I/O commands to OPEN our destination TXT file. A little ROUTE command thereafter redirects output to our TXT file. ROUTE is one of the neatest commands in ZBasic. It provides incredible flexibility. If you wanted to turn this whole program into an AWP file display routine, just ditch the ZBasic OPEN command and reROUTE output to the screen (with a zero).

```
BYTECOUNT = BUFFERSTART
:
ZNUM = 2          : REM opening our TXT file using standard ZBasic I/O
OPEN "O",ZNUM,TXTPATH$ : REM all subsequent PRINTs go to this file
ROUTE ZNUM
:
REM if file is smaller than buffer, make BUFFEREND = end of file
IF AWPLEN# < AMOUNT THEN BUFFEREND  = BUFFERSTART  + AWPLEN#
:
```

CalcLoop, our next main segment, provides most the action in the whole program. It digs out each line record flag (B000 and B001, kinda like the syntax Lissner used), and branches to "DigOutText" if we encounter a line record with ASCII characters in it. Take note that CalcLoop terminates when BYTECOUNT is two less than the end of the buffer. This is because we don't want to grab line record flags that point to text that is not in RAM.

After looping through the buffer, we then test to see if there is more file to read. If so, we pull it in and go back to CalcLoop.

```
"CalcLoop"
DO
  B000 = PEEK(BYTECOUNT)      :REM nab first line record flag (+000)
  IF B000 = 255 THEN "Close"  :REM 255 marks end of file
  B001 = PEEK(BYTECOUNT+1)    :REM now get second flag (+001)
  IF B001 = 0 THEN GOSUB "DigOutText"
  IF B001 = 208 THEN PRINT
  BYTECOUNT = BYTECOUNT + 2
UNTIL BYTECOUNT => BUFFEREND  - 4 : REM don't want to get crossed up at end
of buffer
:
LONG IF TOTALREAD# < AWPLEN#
  TOTALREAD# = TOTALREAD# - 2  : REM need to regrab last two bytes
  FN SET_MARK (OUR_REF,TOTALREAD#) :REM move MARK back by two
  GOSUB "DoRead"
  BYTECOUNT = BUFFERSTART
  GOTO "CalcLoop"
END IF
:
"Close"
FN CLOSE_FILE (0)       :REM close up our files (zero closes all)
:
ROUTE 0                 :REM return output to screen
PRINT
PRINT CHR$(7);"Finished#"
```

```
 END
 :
```

We didn't do CalcLoop all the way through the buffer because of a very subtle I encountered while typesetting this very article! If B001 is a zero, we automatically goto to the DigOutText subroutine. Once there, the routine automatically looks up the next flags - not checking to see if they are actually in memory or not. There are several different ways to handle this. I chose to avoid the problem altogether by backing up the MARK and always getting the last two bytes of the buffer again, thereby making them first on the next pass.

The DigOutText subroutine is only called if the line record flags examined by CalcLoop determine that there is ASCII text data following. DigOutText grabs the fourth line record flag (remember that we skip the third) and figures out the number of text bytes to follow.

```
 "DigOutText"
 B003 = PEEK(BYTECOUNT + 3) : REM skip +002 and get fourth flag
 :
 LONG IF B003 > 128         : REM end of paragraph
   CRFLAG = 1
   B003 = B003 - 128        : REM mask off high bit to get actual chars on
line
 END IF
 :
 TEXTSTART  = BYTECOUNT + 4
 TEXTEND  = TEXTSTART  + B003 - 1   : REM mark beginning and end of ASCII in
memory
 :
 LONG IF TEXTEND  > BUFFEREND    : REM is some of our text not in buffer?
   TOTALREAD# = TOTALREAD# - (AMOUNT - (BYTECOUNT - BUFFERSTART )) : REM
adjust ProDOS MARK to beginning of current line record
   FN SET_MARK (OUR_REF, TOTALREAD#)
   GOTO "Exit"
 END IF
 :
 FOR X  = TEXTSTART  TO TEXTEND    :REM whole line in buffer
   CHAR = PEEK(X)
   :
   REM screen out embedded control characters
   IF CHAR > 31 AND CHAR < 127 THEN PRINT CHR$(CHAR);
   :
 NEXT
 :
 LONG IF CRFLAG = 1         : REM end of paragraph?
   CRFLAG = 0               : REM clear flag
   PRINT                    : REM tack on carriage return
 END IF
 :
 BYTECOUNT = BYTECOUNT + 2 + B003 : REM update bytecount (code above adds 2
more)
 "Exit"
 RETURN                     : REM end of "DigOutText" subroutine
 :
 :
 "DoRead"
```

```
 HOWMUCH = FN READ_FILE (OUR_REF, BUFFERSTART, AMOUNT)
 TOTALREAD# = TOTALREAD# + HOWMUCH     :REM running count of amount of file
read in
 RETURN
 :
 "DiskError"
 FN CLOSE_FILE (0) : REM a zero closes everything
 PRINT:PRINT"Dead due to disk error #:"ERROR
 PRINT ERRMSG$(ERROR)
 STOP
```

The DoRead subroutine calls the READ_FILE function and keeps a running total of the number of bytes actually READ. TOTALREAD# is set to 300 initially because we have skipped the 300 byte AWP header. It is then updated with each READ to reflect the amount of the file already captured.

As I mentioned earlier, the error handling routine is pretty unsophisticated. Even so, it is a good idea to check for errors after every ProDOS MLI call. ZBasic's ONERR GOSUB statement will corral any errors anytime they occur.

We've covered a lot of ground. This fairly short program makes use of a lot of little ZBasic tidbits. It creates a large data buffer using an integer array, it manipulates the ProDOS MLI interface, and uses ZBasic's ROUTE command to full advantage. I hope the program and techniques end up being useful to you.

Znews

Subscription prices in US dollars effective April 15, 1989:
1 yr..$35, 2 yrs..$65 CAN & MEX add $5, other non-USA add $10

Back issues are available at $3.00 each.

Please direct all correspondence to:

Ariel Publishing, Inc.
P.O. Box 266
Unalakaleet, Alaska 99684 USA

Znews is a product of the United States of America.
ZBasic is a registered trademark of Zedcor, Inc.

BULK RATE
U.S. POSTAGE
PAID
SEATTLE, WASH.
PERMIT NO. 74

Mail to:

ARIEL PUBLISHING
P. O. Box 266
Unalakleet, AK 99684
(907) 624-3161

TO: